Paper 298

Obtaining Response Time and Service Level Information About Your SAS® System Applications

Terry Lewis, SAS Institute Inc., Cary, NC
Dan Squillace, SAS Institute Inc., Cary, NC

ABSTRACT

Management and tuning of business transactions and applications has historically been difficult, particularly in distributed environments. Version 2.0 of the ARM API provides a standardized method of instrumenting applications so that variety of performance, availability, and throughput metrics can be monitored. This paper discusses how the ARM API has been implemented in SAS software to enable the instrumentation of business applications. Details on the interface, instrumentation examples, and management reports using the data are discussed.

PREREQUISITES

This paper assumes

- some basic knowledge of application performance and tuning concepts, and
- some general familiarity with SAS Software, e.g., DATA steps and macros.

INTRODUCTION

The measurement of the performance of computer applications has long been a difficult task for the application developer, system administrator, and/or end user. Measuring CPU, network, disk, or other more tangible resources has historically been far easier. Typically, application performance has been measured by one of the following methods:

- Using a performance monitor that can monitor the end-to-end response time of a particular application without any modification of the application itself. Typically, these monitoring tools use very specific operating system or network interfaces to gather communication sequences between the application and the end user, e.g., keyboard lock/unlock, network send/receive, window title capture, packet analysis, etc.
- Modifying the application itself to call functions or subroutines at specific points. The functions typically log the time of the call and other related data to a file. Measuring the time between these function calls yields an approximate response time measurement. This is frequently known as **instrumenting** the application.

At first glance, it would seem that the first technique is superior due to the fact that it requires no alteration of the application. And indeed, it has worked reasonably well in many host-centric environments. However, it also has a number of drawbacks. One of the primary disadvantages is that it frequently is unable to accurately monitor the performance of distributed or client/server applications. Since distributed applications pass both computations and data across many, possibly heterogeneous, computing platforms, the monitor usually only knows about activity on one platform. This provides an incomplete picture since the response time cannot be decomposed into sub-transactions. Attempting to solve this problem by correlating transactions from different application monitors running on different platforms is a difficult task, due to data format differences, time synchronization problems, etc.

Additionally, the monitoring method is frequently unable to ascertain the response time of true **business transactions**. One business transaction may spawn multiples of related sub-transactions that are perceived by the application monitoring software as separate transactions. Since monitoring products do not know anything about the business logic of the application, they cannot accurately report performance metrics that are relevant from a business perspective.

For these and other reasons, applications developers have frequently relied on the second method, instrumentation. While instrumentation has the drawback of requiring alteration of the application, it has numerous advantages. The applications developer knows exactly where transactions start and end from a true business perspective. Correlation of related transactions, possibly running on different computing platforms, is much easier. Additionally, the applications developer is not restricted to only the data that a response time monitor may provide. Instrumenting an application permits the applications developer to log any other kind of business data that they deem relevant.

THE ARM WORKING GROUP AND SAS INSTITUTE

Prior to 1996, application developers instrumented applications via their own proprietary techniques, which made it impossible for systems management vendors to exploit this data. In June 1996, Hewlett-Packard and Tivoli Systems announced a collaboration to develop an open, vendor-neutral approach to manage the performance of distributed applications. They were joined by other software vendors, including SAS Institute, as well as interested end-user companies, to form an industry partnership known as the **Application Response Measurement (ARM) Working Group**. The ARM API standard was the result of this effort.

SAS Institute develops tools for application development as well as packaged or customized applications for specific business purposes. Additionally, SAS Institute provides IT Service Vision™ Software, a solution for performance management that addresses an enterprise's entire spectrum of services including systems, networks, Web servers and phone systems. Performance data is warehoused and analyzed to provide reporting for all levels of an IT organization and to help integrate IT goals with corporate business goals.

One of our primary objectives in joining the ARM Working Group was to provide input to the group so that we could easily develop our own application performance tools based on the ARM API. We felt that our customers would find these tools to be a significant added value when developing their own applications using SAS Software. We also hoped to use these tools ourselves when instrumenting our own packaged business applications. In addition, we felt that the data logged by the ARM API would be a valuable data source for our IT Service Vision software, which would provide our customers with automated facilities for historical service level and response time reporting.

THE ARM API

The ARM API is a standard set of function calls that are callable from your application program. Typically, you embed these

function calls at strategic points within your application so that they collect the desired transaction response time or other data that you wish to gather.

The ARM API was primarily implemented as a C/C++ based API, although it is callable from a wide variety of other languages. Other language implementations, such as Java, are currently under development by the ARM Working Group.

If you desire to instrument your own applications and/or software, an ARM Software Development Kit (SDK) is available free from the ARM Working Group at

http://www.cmg.org/regions/cmgarmw/index.html

This kit contains source code, header files, examples, and a sample logger.

How does the ARM API work? The first, and most important, step is to identify the key business transactions in your application. You should choose transactions that are performance sensitive and for which corrective action can be taken if their response time is too slow. In particular, transactions that are highly visible to the end user are good candidates.

Once the transactions of interest have been identified, you then modify the application, embedding the ARM calls at strategic points. The typical usage is to place an **arm_start** call just before the transaction is initiated to signify the beginning of the transaction. An associated **arm_stop** call would be placed in the application code where the transaction is known to be complete.

When using the ARM API in C application programs, those programs must be linked so that the ARM function calls are resolved. The ARM calls are resolved by the existence, in the standard link search paths, of an executable image known as the **ARM Shared Library Module**. The Shared Library module defines the ARM functions so that the external references are resolved. This executable is implemented as a DLL on Windows and OS/2, a shared library on HP-UX, and a shared object on Sun Solaris. The name of the Shared Library module is platform-specific, but always begins with the letters **"libarm"**.

For production usage, you normally link your application program with a Shared Library module supplied by your systems management vendor. This Shared Library module should be located in a common system library that is accessible during program linkage. Hewlett-Packard, Tivoli, and other systems management vendors supply their own Shared Library modules. The responsibility of the Shared Library module is to accept the ARM function parameters from the calling application program and log that information persistently. It can log to a file or database synchronously in the same process or session, or it can route the information over a communication link to a **measurement agent** running in a separate process that logs the information asynchronously. The Measurement Agent centralizes management of the ARM function calls and could possibly display the response time information on a monitor or console in visual form, as well as write the information to a log file.

The ARM SDK also supplies a Shared Library module known as the **NULL Shared Library** that can be used for testing an instrumented application. It simply defines each ARM function and returns control immediately back to the calling program without performing any logging or other work.

Version 1 of the ARM API was announced in June 1996. Version 2 of the ARM API, released November 1997, established definitions of previously reserved parameters that can optionally be used for:

- correlating related transactions in a parent/child manner so that transactions in the client/server model can be associated, and
- providing the ability for the application to log metrics via simple counters, gauges and strings to supplement the response time data.

The six ARM API function calls are briefly described below -- see the *ARM API Guide, Version 2 for* a more complete description of the functions and their parameters.

arm_init

Initializes the ARM environment and names the application and optionally the user of the application. A unique identifier **(application id)** is returned to the calling program. Example:

```
my_appl_id = arm_init ("My application",
             "userxyz", 0, 0, 0);
```

arm_getid

Names a **transaction class**. Transaction classes are related units of work within an application. A unique identifier **(transaction class id)** is returned to the calling program. Example:

```
txn_clsid = arm_getid (my_appl_id,
            "DW Queries",
            "Query Customer Database",
            0, buffer_ptr, buffer_len);
```

The `buffer_ptr` and `buffer_len` parameters point to an optional buffer supplied by the calling program that contains a template, i.e., metadata, that describes data that will be logged in subsequent arm_start, arm_update, or arm_stop calls. The type of metadata and data, format of the buffer, etc., is explicitly documented in the ARM API Guide.

arm_start

Signals the start of a transaction. A unique identifier **(start handle)** is returned to the calling program. Example:

```
shdl = arm_start (txn_clsid, 0,
       0, buffer_ptr, buffer_len);
```

The optional `buffer_ptr` and `buffer_len` parameters point to any data described by metadata from a previous arm_getid call.

arm_update

An optional function that can be used between an arm_start and arm_stop to supply any information about a transaction in progress. A return code is returned to the calling program. Example:

```
rc = arm_update (shdl, 0, buffer_ptr,
     buffer_len);
```

As with the arm_start call, the `buffer_ptr` and `buffer_len` parameters point to a buffer supplied by the calling program that contains additional information that is to be logged.

An additional feature of the arm_update call is that it can supply not only the highly formatted buffer (known as a **Format 1 Buffer)** that arm_start and arm_stop uses, but can also pass an unformatted buffer (known as a **Format 2 Buffer)** containing up to 1020 bytes of data.

arm_stop

Signals the end of a transaction. An optional transaction status code may be supplied. A return code is returned to the calling program. Example:

```
rc = arm_stop (shdl, ARM_GOOD,
     0, buffer_ptr, buffer_len);
```

arm_end

Terminates the ARM environment and signals the end of an application. It's primary purpose is to free any memory associated with the application.

```
rc = arm_end (my_appl_id, 0, 0, 0);
```

THE SAS INTERFACE TO ARM

As stated earlier, one of our primary objectives at SAS Institute was to adapt the ARM API model for use in SAS applications. While SAS Institute develops and markets many packaged applications for specific purposes, it has long been known as a supplier of generic tools for data analysis, application development, data warehousing, and statistical processing. Customers have used these tools in combination to design and write their own applications. We felt that the best design would be to supply ARM functionality via the same toolkit approach using familiar SAS constructs so that SAS application developers could easily measure the response time of their applications using techniques with which they were comfortable. Those tools could be used by both our internal software developers responsible for SAS Institute-packaged applications, as well as by customers for use in their own custom applications.

Another design goal was to make the interface simple to use and unobtrusive as possible. Since the primary drawback of instrumentation is that existing applications must modified (thus incurring additional testing), it was our desire that insertion of ARM calls in SAS applications should, as much as possible, avoid perturbing any code surrounding the call.

We decided that the SAS Interface to ARM would have four distinct interfaces within SAS software:

- A C function layer,
- A set of SAS DATA Step/SCL functions,
- A set of SAS macros,
- A set of SAS/AF® classes and methods.

The C function interface was implemented via development of an ARM Shared Library module that strictly adheres to the ARM API standard. It is primarily for use by internal, C-based SAS Institute products so is not described in this paper.

The SAS/SCL function interface consists of six SAS functions that closely correspond to the six ARM functions. All SAS functions can be used in the SAS DATA step, or in SAS Screen Control Language (SCL), a language used to build interactive applications. While the syntax of the SAS ARM functions is very similar to the ARM API C functions, minor modifications were necessary due to language differences.

Internally, the SAS functions themselves are written in C. When a function is called from a SAS application program the first time, it searches the standard search paths and dynamically loads the first Shared Library module it finds. The Shared Library module can be one supplied by a systems management vendor or can be the SAS Institute-supplied one. The decision of which one to use is made when configuring the SAS Interface to ARM.

The SAS macro interface consists of six macros that correspond to the six ARM functions. The macros "wrap" the SAS functions and add additional capabilities not available to the SAS functions, as well as relieve the applications developer from many housekeeping chores associated with using the functions. The macros are the recommended interface for most SAS applications development.

The object-oriented interface, implemented as a set of SAS/AF classes and methods, is primarily designed for the SCL programmer who prefers working in an object-oriented environment. The object-oriented interface is functionally identical to the macro interface.

The following sections describe the various interfaces to ARM in more detail.

THE FUNCTION INTERFACE FOR ARM

All functions for the SAS Interface to ARM begin with the letters **"trans"**, short for "transaction". Optional function parameters are denoted with <>.

Trans_app_init

The trans_app_init function corresponds to the arm_init function and is used to initialize the ARM environment and name the application.

Syntax:

```
app_id = trans_app_init( app_name
         <,app_user> );
```

Trans_getid

The trans_getid function corresponds to the arm_getid function and is used to name a transaction class.

Syntax:

```
txn_id = trans_getid( app_id,
         txn_name <,txn_detl> );
```

Trans_start

The trans_start function corresponds to the arm_start function and is used to signify the start of a transaction.

Syntax:

```
shandle = trans_start( txn_id );
```

Trans_update

The trans_update function is an optional call that corresponds to the arm_update function. It can be used to show the progress of a long transaction.

Syntax:

```
rc  = trans_update( shandle,
      <,data> );
```

Trans_stop

The trans_stop function signifies the end of a transaction started by trans_start and corresponds to the arm_stop function. An optional transaction status can be passed that indicates the success or failure of the transaction.

Syntax:

```
rc = trans_stop( txn_id <,txn_stat> );
```

Trans_end

The trans_end function signifies the end of the application and corresponds to the arm_end function.

Syntax:

```
rc = trans_end( app_id );
```

Figure 1 is a simple DATA step example utilizing all of the ARM functions:

```
data _null_;
  rc = 0;

  /*  Initialize the application  */
  appl_id = trans_app_init( "Appl 1",
            "&sysjobid" );

  /*  Get a transaction class id */
  txn_id = trans_getid( appl_id, "Txn 1",
           "Transaction #1 detail" );

  /*  Note the beginning of a transaction */
  sHandle = trans_start( txn_id );

  /* The transaction starts here.  Issue  */
  /* trans_updates as needed during the   */
  /* transaction                          */
  do i = 1 to 100000;
    if mod(i, 10000) = 0 then
      rc = trans_update( sHandle,
                         "loop count=" || i
                       );
  end;
  txn_stat = rc;

  /*  Note the end of the transaction */
  rc = trans_stop( sHandle, txn_stat );

  /*  End the application */
  rc = trans_app_end( appl_id );

run;
```

Figure 1 - ARM Functions

Note that while this example shows all the ARM function calls being performed in the same DATA step, you can place them in separate DATA steps (or separate SCL/FRAME programs) as well. However, placing them in separate DATA steps complicates keeping track of the ids (application ids, transaction class ids, and start handles) returned from the ARM function calls, since SAS DATA step variables only exist for the life of the DATA step. The reality is that almost all SAS applications would utilize the ARM macros in this manner, which would necessitate a greater programming burden on the application programmer.

However, the SAS Macro Interface, described below, greatly simplify the management of those ids.

THE MACRO INTERFACE FOR ARM

For each ARM Function, there exists a corresponding SAS macro that "wraps" the SAS ARM function and provides additional capabilities. The following table shows the relationship:

ARM API C functions	SAS ARM functions	SAS ARM macros
arm_init	trans_app_init	%arminit
arm_getid	trans_getid	%armgtid
arm_start	trans_start	%armstrt
arm_update	trans_update	%armupdt
arm_stop	trans_stop	%armstop
arm_end	trans_end	%armend

Figure 2 - Function/Macro Relationship

Figure 3 produces the same results as Figure 1 shown previously:

```
data _null_;

  /*  Initialize the application  */
  %arminit(appname="Appl 1",
           appuser="&sysjobid");

  /*  Get a transaction class id */
  %armgtid(txnname="Txn 1",
           txndet="Transaction #1 detail" );

  /*  Note the beginning of a transaction */
  %armstrt;

  /* The transaction starts here.  Issue  */
  /* trans_updates as needed during the   */
  /* transaction                          */
  do i = 1 to 100000;
    if mod(i, 10000) = 0 then
      %armupdt(data="loop count=" || i );
  end;
  txn_stat = _armrc;

  /*  Note the end of the transaction */
  %armstop(status=txn_stat);

  /*  End the application */
  %armend;

run;
```

Figure 3 - ARM Macros

As Figure 3 illustrates, the manual passing of ids is unnecessary in the application program. The macros transparently handle the passing of ids back and forth between function calls, even between DATA steps. All macro parameters are keyword parameters and correspond to the ARM function parameters.

The general rule is when two or more ARM macros are coded in a single DATA step, then the ARM macro uses DATA step variables to pass the values of the ids. DROP statements are generated so that these variables are not inadvertently kept in an output dataset. When ARM macros appear on separate DATA steps, then the ARM macros use global macro variables to pass the value of the ids.

The following table describes the variables (both DATA step and global macro) that are used to pass ids between macros:

variable	description	set by	used by
_armapid	application id	%arminit	%armend
_armtxid	transaction id	%armgtid	%armstrt
_armshdl	start handle	%armstrt	%armstop, %armupdt
_armrc	error status	%armupdt, %armstop, %armend	none

Figure 4 - Global ARM Variables

Because of possible conflicts, you should avoid naming any of your own variables beginning with the letters **_arm**.

While this technique works well in basic scenarios like Figure 3, concurrent applications, transaction classes, or transaction instances can pose problems:

```
data _null_;
  %arminit(appname="Appl 1",
           appuser="&sysjobid");
  %armgtid(txnname="Txn 1",
           txndet="Transaction class 1" );
  %armgtid(txnname="Txn 2",
           txndet="Transaction class 2" );
run;

data _null_;
  /* Start an instance of transaction  */
  /* class 1                           */
  %armstrt;
run;
.
.
.
```

Figure 5 - Concurrent Transactions - Wrong!

The comments indicate that the programmer thinks that the %armstrt in the last DATA step is starting an instance of transaction class 1. However, since the most recent %armgtid was for transaction class 2, the %armstrt will actually start an instance for it, since the _armtxid global macro variable contains the most recent transaction class id.

To avoid this problem, you can specify your own uniquely named variables to contain the id value to be passed to other DATA steps or SCL programs via the ***var** parameters, i.e., **appidvar**, **txnidvar**, and **shdlvar**. This facility gives you complete control over concurrently active applications and transactions. The following example demonstrates concurrent transaction classes and instances:

```
data _null_;
  %arminit(appname="Appl 1",
           appuser="&sysjobid");
  %armgtid(txnname="Txn 1",
           txndet="Transaction class 1",
           txnidvar=txncls1 );
  %armgtid(txnname="Txn 2",
           txndet="Transaction class 2",
           txnidvar=txncls2 );
run;

data _null_;
  /* This time, point to the correct   */
  /* transaction class.  Save the      */
  /* start handle as well.             */
  %armstrt(txnidvar=txncls1,shdlvar=sh1);
run;

data _null_;
  /* Same thing here using             */
  /* transaction class 2.              */
  %armstrt(txnidvar=txncls2,shdlvar=sh2);
run;

data _null_;
  /* Now point to the desired          */
  /* transaction we want to update.    */
  %armupdt(data="Updating txn 1 ...",
           shdlvar=sh1);
run;

data _null_;
  %armstop(shdlvar=sh2);
  %armstop(shdlvar=sh1);
  %armend;
run;
```

Figure 6 - Concurrent Transactions - Correct!

Using the *var parameters makes instrumentation of existing applications very easy. No logic changes to existing programs are required -- just insert the proper macros at the desired transaction points.

Another additional feature of the macros is conditional execution. It is frequently desirable to instrument an application with ARM calls, however, have the ability to disable all calls by default, or to only turn on selected sets of ARM calls.

The **level** parameter on the macros allows you to identify an ARM macro with a particular **execution level**. Whether or not that macro executes then depends on the setting of two global macro variables, **_armglvl**, and **_armtlvl**. _armglvl is the **global level** macro variable. If the value of the level parameter on the macro is less than or equal to _armglvl, then the macro will execute:

```
/* set the global level to 1 */
%let _armglvl = 1;

data _null_;
  %arminit(appname="Appl 1",
           appuser="&sysjobid");
  %armgtid(txnname="Txn 1",
           txndet="Transaction #1 detail");
run;

data _null_;
  /* level 1 macros - these will  */
  /* execute.                     */
  %armstrt(level=1);
  %armstop(level=1);

  /* level 2 macros - these will NOT */
  /* execute.                        */
  %armstrt(level=2);
  %armstop(level=2);

  /* no level parm - this will execute */
  %armend;
run;
```

Figure 7 - Use of _armglvl

_armtlvl is the **target level** macro variable is similar to _armglvl except the macro will only execute if it equal to the target level:

```
/* set the target level to 3 */
%let _armtlvl = 3;

data _null_;
  %arminit(appname="Appl 1",
           appuser="&sysjobid");
  %armgtid(txnname="Txn 1",
           txndet="Transaction #1 detail");
run;

data _null_;
  /* level 3 macros - these will */
  /* execute.                    */
  %armstrt(level=3);
  %armstop(level=3);

  /* level 1 macros - these will NOT */
  /* execute.                        */
  %armstrt(level=1);
  %armstop(level=1);

  /* no level parm - this will execute */
  %armend;
run;
```

Figure 8 - Use of _armtlvl

If you set both _armglvl and _armtlvl at the same time, then both values are compared to determine whether the macro should be executed or not.

Another macro variable, _**armexec**, is available to globally disable the ARM macros, no matter what the setting of _armglvl or _armtlvl. Set _armexec to zero to disable all macros, and set it to any other value to re-enable all macros.

Using the level parameter in conjunction with the conditional execution macro variables allows you to establish hierarchies of ARM calls in your application programs and permits complete flexibility with respect to what calls are logged.

THE OBJECT-ORIENTED INTERFACE FOR ARM

In addition to the function and macro interfaces, an object-oriented interface is also supplied. This interface is patterned after the macro interface and was developed for those application programmers that prefer to work in an object-oriented environment.

The object-oriented interface consists of four classes:

- ARMOBJ - The ARM Object parent class
- ARMAPP - The ARM Application class
- ARMTXNC - The ARM Transaction Class class
- ARMTXNI - The ARM Transaction Instance class

ARMAPP.CLASS , ARMTXNC.CLASS, and ARMTXNI.CLASS are subclasses of ARMOBJ.

To issue ARM calls in the object-oriented interface, methods are sent to ARM objects. These methods are analogous to the macros and functions discussed previously. The following table shows the relationship between the C functions, macros and object-oriented interfaces with respect to the six ARM API calls:

ARM API C functions	SAS ARM macros	SAS ARM methods
arm_init	%arminit	_app_init_
arm_getid	%armgtid	_getid_
arm_start	%armstrt	_start_
arm_update	%armupdt	_update_
arm_stop	%armstop	_stop_
arm_end	%armend	_app_end_

Figure 9 - Function/Macro/Method relationship

Instead of passing values on parameters as is done in the function or macro interface, methods are issued that initialize instance variables in instantiated objects. The following SCL is a basic example of how the object-oriented interface can be used:

```
/*
 *  Init the application
 */
 i_armapp = instance(
            loadclass('armapp.class'));

 appname = 'Example App 1';
 appuser = 'userxyz';
 call send( i_armapp, '_SET_APP_NAME_',
            appname );
 call send( i_armapp, '_SET_APP_USER_',
            appuser );
 call send( i_armapp, '_APP_INIT_',
            appid );

/*
 *  Define a transaction class
 */
 i_armtxc = instance(
            loadclass('armtxnc.class'));

 txcname = 'Txn 1';
 txcdet  = 'Example Txn Class 1';
 call send( i_armtxc, '_SET_APP_ID_',
            appid );
 call send( i_armtxc, '_SET_TXN_NAME_',
    txcname );
 call send( i_armtxc, '_SET_TXN_DETAIL_',
            txcdet );
 call send( i_armtxc, '_GETID_', txcid );

/*
 *  Start a transaction instance
 */
 i_armtxi = instance(
            loadclass('armtxni.class'));
 call send( i_armtxi, '_SET_TXN_ID_',
            txcid );
 call send( i_armtxi, '_START_', shandle );

/*
 *  Update a transaction instance
 */
 data  = 'format 2 data buffer';
 call send( i_armtxi, '_SET_DATA_', data );
 call send( i_armtxi, '_UPDATE_', rc );

/*
 *  Stop a transaction instance
 */
 status = 1;
 call send( i_armtxi, '_SET_STATUS_', status );
 call send( i_armtxi, '_STOP_', rc );
 call send( i_armtxi, '_TERM_' );

/*
 *  Stop the application
 */
 call send( i_armapp, '_APP_END_', rc );
 call send( i_armtxc, '_TERM_' );
 call send( i_armapp, '_TERM_' );
```

Figure 10 - Object-Oriented Interface to ARM

LOGGING

As described earlier, the first ARM call results in a search of the standard search paths for an ARM Shared Library module. The responsibility of this module is to accept the function call parameters, perform any error checking, initiate the logging of the data associated with the ARM function, and return control back to the application program with an appropriate id or error status.

Some sites will prefer to use a Shared Library module supplied by a systems management vendor so that application metrics from ARM API calls are integrated with the rest of their performance data. Frequently, logging of ARM data may be performed by a Measurement Agent running in a separate process that is listening on a communications port for the ARM data. This has the benefit of centralizing all ARM data into a single location for logging and post-processing.

If you choose not to use another vendor's Shared Library module, the SAS Interface to ARM supplies a basic logger that will capture response time and CPU time statistics and handle the task of logging this information persistently. By default, the SAS Logger runs in the same session as the client application and logs all ARM data synchronously to the SAS Log. However, if you assign an external file (via the **filename** statement) with the fileref of **ARMLOG**, then all ARM calls are routed to the file pointed to by the fileref. This example logs the ARM calls to a text file on the Windows operating system:

```
filename armlog 'c:\temp\armlog.txt';
```

```
data _null_;
  %arminit(appname="Appl 1",
           appuser="&sysjobid");
  %armgtid(txnname="Txn 1",
           txndet="Transaction #1 detail" );
run;
```

Figure 11 - Logging to an External File

To resume logging to the SAS log, deassign the fileref.

This local, synchronous mechanism for logging using the SAS Logger is ideal when you would rather log on an as-needed basis, rather than continuously. For example, an application could be instrumented with ARM calls that are disabled by default and only enabled when problems are noticed. Logging at the local machine would allow for convenient, easy analysis of data that is specific to the individual problem and user.

In Version 6.11 of the SAS System, the filename statement has been enhanced to point to other types of files rather than plain flat files. For example, assigning the filename statement to a socket could provide a mechanism to gather the ARM calls to a remote central location.

The format of the ARM log written to by the SAS Logger is primarily designed to be easily readable with the naked eye. The date/time stamp and the call identifier always appear in the same column location. Subsequent information appears as a name=value pair. An example of the ARM log is appears in Figure 16 at the end of this paper. All information passed by the application program on its ARM API calls is written to the log, as well as other calculated statistics. Some of the information written to the log is listed below in Figure 12:

name	description
AppAvgRt	Average response time for all transactions in the application
AppCpu	CPU time for the entire application
AppElap	Elapsed time for the entire application
AppID	Application ID
AppMaxTx	Longest transaction in the entire application
AppMinTx	Shorted transaction in the entire application
AppName	Application name
AppNCls	Number of transaction classes in the application
AppNTxSp	Number of transactions in the application that were stopped
AppNTxSt	Number of transactions in the application that were started
AppNTxUp	Number of transactions in the application that were updated
AppUser	Application UserID
ClsID	Transaction class ID
TxCpu	CPU time for a transaction
TxDet	Transaction class detail
TxElap	Elapsed time for a transaction
TxName	Transaction class name
TxShdl	Transaction instance start handle
TxStat	Transaction status code
UpdtData	Additional application-supplied data on TRANS_UPDATE calls

Figure 12 - Log Variables

REPORTING

To process the ARM log data programmatically, a macro, **%ARMPROC**, has been supplied that will read and process the information from the log file into six SAS datasets. The six datasets correspond to the six ARM calls and are named similarly. For example,

```
%armproc( log=c:\arm\logfile, lib=WORK );
```

will process the ARM log data in `c:\arm\logfile` and write the six SAS datasets to the WORK library. Once the six datasets are available, an additional macro, **%ARMJOIN**, may be used to merge information about applications and transactions. This macro creates an APP dataset containing information about all applications found in the log file, and one or more transaction datasets. The transaction datasets are named TXNxxxxx where xxxxx is a numeric identifier that is incremented sequentially for each application that is encountered, for example, TXN1, TXN2, etc.

Using these datasets, you can create a multitude of reports, charts, or graphs that visually depict response time or CPU time information for applications or transactions. A sample SAS/AF application has been developed that will process the ARM logs and visually display the data using the Graphics Object. This application will be packaged with the SAS Interface to ARM in the Version 8 release of the SAS System.

Additionally, a **Generic Collector** is under development for IT Service Vision Software that will enable the processing of data residing in SAS ARM Logs. This will allow the ARM data to be periodically warehoused and easily reported on by robust IT Service Vision facilities.

INCORPORATING ARM MEASUREMENTS INTO APPLICATION DEVELOPMENT AND MANAGEMENT STRATEGIES

ARM instrumentation can play a key role in both the development and production management phases of new applications. During the development phase, it can be used to spot performance and scalability issues early enough to implement cost-effective remedies. Making design and architectural changes to address performance issues late in the development cycle is extremely costly in both time and resources and can ultimately result in the project failing. ARM instrumentation has also been shown to be an effective application tracing tool which can significantly reduce debugging time by providing a log of events leading up to the failure. Furthermore, detailed transaction tracing can be carried out by embedding transaction UPDATE calls at points of interest within the transaction.

Once an application is put into production, data gathered through the ARM interface can be used to track response times, generate usage data for performance reporting, charge back, and capacity planning. In addition, the ARM interface now gives distributed systems applications programmers a platform-independent way to communicate application alerts to operations consoles. Currently, there is no way for an application running on a client to signal a central operations center that , for example, the client was unable to open a particular remote data base.

To help manage these varied uses of ARM effectively, we recommend that the granularity of ARM logging be controlled by one of the conditional execution macro variables and that specific ARM calls are made depending on the value of that variable. In this way, many ARM API calls could be embedded in application programs but only a subset of these may be issued depending on the setting of the variable. For example, assume that the global level variable _ARMGLVL could be set with one of the following values:

1 = calls associated with alerts
2 = calls associated with basic transaction start/end recording
3 = transaction update calls needed for detailed transaction tracing.

In this case, a setting of 1 would result in only alerts being generated through the ARM interface, a setting of 2 would result in alerts and basic transaction information, and a setting of 3 would result in all ARM API calls being executed. Furthermore, if a line command or menu option was implemented to allow dynamic setting of the global level variable, then the level could be altered as needed. For example, assume a user is having problems and calls the Help Desk for assistance. The Help Desk may determine that more diagnostic information would be helpful and request the user to reset the global level variable to gather the more detailed trace information.

ARM can also be used to define and report on a series of transactions that taken together constitute a business process. Consider a customer interaction with an ATM. The period of time from the moment the bank card is inserted into the machine until the time it is removed could be considered a "customer service process". The transactions which occur during this time period are all elements of the process (e.g. withdrawals, deposits, balance checks). Furthermore, the time intervals between transactions may be considered the manual part of the business process and therefore are as important to track and measure as the individual transactions.

USING THE ARM API INTERNALLY AT SAS INSTITUTE

At SAS Institute, most internal applications are written in SAS. When employees in one of our departments complained of poor response time displaying windows in one of their SAS/AF applications, we saw this as an opportunity to use the SAS Interface to ARM as a diagnostic tool.

The best method of instrumenting an existing application is to first determine where, from the user's perspective, the problems are occuring. Once that is determined, inserting a few basic ARM API calls at those critical points in the code is all that usually needed in the first iteration of instrumentation. The temptation to "micro-instrument" every subroutine or method call should be initially avoided so that manageable amounts of performance data can be easily produced and analyzed.

To get started, we inserted the necessary %ARMINIT and %ARMGTID calls near the beginning of the application as well as set up the ARM log file and any global macro variables that are required:

```
call symput('_armexec', '1');
call symput('_armglvl', '1');

rc = filename( 'armlog',
               'c:\intapp\ARM\logs\test1.txt'
             );

%arminit( appname='Internal SAS Application',
          appuser="&sysjobid",
          appidvar=appid1,
          level=1,
          scl=yes
        );

%armgtid( appidvar=myappid,
          txnname='Window',
          txndet='Window display stats',
          txnidvar=txnid1,
          level=1,
          scl=yes
        );
```

Figure 13 - Initialization

To track the response time when displaying new SAS/AF frames, an %ARMSTRT macro was inserted in the SCL for the calling frame prior to any CALL DISPLAY statements. An %ARMUPDT macro was also used to log supplemental information about where we were within the application:

```
.
.
.
%armstrt( txnidvar=txnid1,
          shdlvar=shact1,
          level=1,
          scl=yes
        );
%armupdt( shdlvar=shact1,
          level=1,
          data='call display window2',
          scl=yes
        );
call display( 'window2.frame' );
.
.
.
```

Figure 14 - Starting the Transaction

To capture the time it took to display the new window, an %ARMSTOP macro was inserted at the end of the INIT section in the SCL of the called frame, e.g.,

```
INIT:
.
.
.
%armstop( shdlvar=shact1,
          level=1,
          scl=yes
        );
return;
```

Figure 15 - Stopping the transaction

Note that the **shdlvar=** parameter points to the same variable on both the %ARMSTRT and %ARMSTOP so that the same start handle is used for both API calls.

This technique captures the time it takes to initialize and display a new window. Once this first iteration of instrumentation was performed, we then decided to instrument further sections of code within the INIT section of the called frame with more %ARMSTRT/%ARMSTOP pairs. These new "sub-transactions" would give us a more complete picture about how the entire end-to-end response time of the window display was composed. On the next iteration, analysis of the data pointed the cause of the problem to a single subroutine that accounted for 88% of the response time of the entire transaction! Plans were made to move the functionality of this subroutine out of the mainline path.

This simple example demonstrates the benefits of using the SAS Interface to ARM as a diagnostic tool. It can also be used in more strategic scenarios as a capacity planning tool to regularly gather application metrics over long periods of time. By capturing and monitoring historical performance data about your applications, you can begin to forecast when future application performance problems may occur.

CONCLUSION

ARM is an important step forward towards fulfilling the need to standardize instrumentation of applications, particularly distributed applications. ARM gives you direct, concrete information about the problems the end user is having, which

enables you to then use that information for guidance and direction when subsequently diagnosing the problem at lower levels, e.g., analyzing reports on server workloads, network traffic, etc.

The SAS Interface to ARM is a simple and non-invasive method of instrumenting SAS applications using the ARM API. It provides, for the first time, a flexible and easy method of interfacing information about SAS application performance directly to systems management products. It also provides logging and reporting facilities that allow it to interface with other Institute products, such as IT Service Vision. Judicious usage of the SAS Interface to ARM will result in increased awareness of application performance and activity, thus giving rise to solutions that will improve end user productivity and satisfaction.

REQUIREMENTS AND PLATFORMS

The SAS Interface to ARM software will run on Windows 3.1, Windows 95, Windows 98, Windows NT, and OS/2 platforms running SAS Version 6.12. The ARM functions and macros require Base SAS software. The object-oriented interface requires SAS/AF software. This version of the software is experimental and is available only upon request.

In 1999, the SAS Interface to ARM will be packaged with the Base SAS product and run on all platforms running SAS Version 8. This includes MVS and various Unix operating systems in addition to the Windows and OS/2 platforms.

REFERENCES

ARM Working Group (1997), *Application Response Measurement 2.0 API Guide,* Roseville, CA and Austin, TX: Hewlett-Packard Company and International Business Machines Inc.

Morris, Denise, "*Managing the Enterprise with the Application Response Measurement API (ARM)"*, Roseville, CA: Hewlett-Packard Company

SAS Institute Inc. (1995), *SAS Software Changes and Enhancements, Release 6.11,* Cary, NC: SAS Institute Inc.

AUTHOR CONTACT

Terry Lewis, SAS Institute Inc., 100 SAS Campus Dr., Cary, NC 27513, (919)677-8000, ext. 7778, email snotll@wnt.sas.com

Dan Squillace, SAS Institute Inc., 100 SAS Campus Dr., Cary, NC 27513, (919)677-8000, ext. 7611, email snodjs@wnt.sas.com

Figure 16 - Sample SAS Log

Some lines have been wrapped to format to the page. Wrapped lines are indented.

```
19JAN1999:15:41:55.171  ARM_INIT    AppID=4  AppName=Appl 1  AppUser=snotll
19JAN1999:15:41:55.171  ARM_GETID   AppID=4  ClsID=4  TxName=CustDB  TxDet=Customer Database
  Queries
19JAN1999:15:41:55.186  ARM_START   AppID=4  ClsID=4  TxSHdl=6
19JAN1999:15:41:58.874  ARM_UPDATE  AppID=4  ClsID=4  TxSHdl=6  TxElap=0:00:03.688
  TxCpu=0:00:03.687  UpdtData=First query still running ...
19JAN1999:15:42:02.546  ARM_UPDATE  AppID=4  ClsID=4  TxSHdl=6  TxElap=0:00:07.360
  TxCpu=0:00:07.359  UpdtData=First query still running ...
19JAN1999:15:42:06.233  ARM_UPDATE  AppID=4  ClsID=4  TxSHdl=6  TxElap=0:00:11.047
  TxCpu=0:00:11.047  UpdtData=First query still running ...
19JAN1999:15:42:09.921  ARM_UPDATE  AppID=4  ClsID=4  TxSHdl=6  TxElap=0:00:14.735
  TxCpu=0:00:14.733  UpdtData=First query still running ...
19JAN1999:15:42:09.921  ARM_STOP    AppID=4  ClsID=4  TxSHdl=6  TxStat=0  TxElap=0:00:14.735
  TxCpu=0:00:14.733
19JAN1999:15:42:09.936  ARM_START   AppID=4  ClsID=4  TxSHdl=7
19JAN1999:15:42:13.624  ARM_UPDATE  AppID=4  ClsID=4  TxSHdl=7  TxElap=0:00:03.688
  TxCpu=0:00:03.671  UpdtData=Second query still running ...
19JAN1999:15:42:17.327  ARM_UPDATE  AppID=4  ClsID=4  TxSHdl=7  TxElap=0:00:07.391
  TxCpu=0:00:07.343  UpdtData=Second query still running ...
19JAN1999:15:42:17.343  ARM_STOP    AppID=4  ClsID=4  TxSHdl=7  TxStat=0  TxElap=0:00:07.407
  TxCpu=0:00:07.359
19JAN1999:15:42:17.343  ARM_END     AppID=4  AppNCls=1  AppNTxSt=2  AppNTxSp=2  AppNTxUp=6
  AppElap=0:00:22.172  AppCpu=0:00:22.125  AppAvgRt=0:00:11.071  AppMinTx=0:00:07.407
  AppMaxTx=0:00:14.735
```